Teaching programming at a distance: the Internet software visualization laboratory

Domingue, J. & Mulholland, P. *Teaching Programming at a Distance: The Internet Software Visualization Laboratory.* Published 8 Oct. 1997
Journal of Interactive Media in Education, 97 (1).
[www-jime.open.ac.uk/97/1] ISSN: 1365-893X

Teaching Programming at a Distance: The Internet Software Visualization Laboratory

John Domingue, Paul Mulholland

Abstract:

This paper describes recent developments in our approach to teaching computer programming in the context of a part-time Masters course taught at a distance. Within our course, students are sent a pack which contains integrated text, software and video course material, using a uniform graphical representation to tell a consistent story of how the programming language works. The students communicate with their tutors over the phone and through surface mail.

Through our empirical studies and experience teaching the course we have identified four current problems: (i) students' difficulty mapping between the graphical representations used in the course and the programs to which they relate, (ii) the lack of a conversational context for tutor help provided over the telephone, (iii) helping students who due to their other commitments tend to study at 'unsociable' hours, and (iv) providing software for the constantly changing and expanding range of platforms and operating systems used by students.

We hope to alleviate these problems through our Internet Software Visualization Laboratory (ISVL), which supports individual exploration, and both synchronous and asynchronous communication. As a single user, students are aided by the extra mappings provided between the graphical representations used in the course and their computer programs, overcoming the problems of the original notation. ISVL can also be used as a synchronous communication medium whereby one of the users (generally the tutor) can provide an annotated demonstration of a program and its execution, a far richer alternative to technical discussions over the telephone. Finally, ISVL can be used to support asynchronous communication, helping students who work at unsociable hours by allowing the tutor to prepare short educational movies for them to view when convenient. The ISVL environment runs on a conventional web browser and is therefore platform independent, has modest hardware and bandwidth requirements, and is easy to distribute and maintain. Our planned experiments with ISVL will allow us to investigate ways in which new technology can be most appropriately applied in the service of distance education.

Keywords: Distance teaching, teaching computer programming, Software Visualization, evaluation, Prolog.

Demonstrations:

The Web version of this article has an interactive demonstration of ISVL embedded in it as a Java applet. (requires a Java-compliant browser).

Commentaries:

All JIME articles are published with links to a commentaries area, which includes part of the article's original review debate. Readers are invited to make use of this resource, and to add their own commentaries. The authors, reviewers, and anyone else who has 'subscribed' to this article via the website will receive email copies of your postings.

J. Domingue & P. Mulholland, *Knowledge Media Institute, The Open University, Walton Hall, Milton Keynes, MK7 6AA U.K. {J.B.Domingue | P.Mulholland}@open.ac.uk.*
http://kmi.open.ac.uk/sv

1. Introduction

Teaching computer science has never been a straightforward or simple process. As a result, a great deal effort has been aimed at improving the teaching process, from intelligent tutoring approaches such as the Lisp Tutor (Anderson and Reiser, 1985), to educational visualizations of programs such as Baecker and Sherman's (1981) "Sorting out sorting" movie. The potential value of effective support tools helping students grasp difficult programming concepts is even greater in a distance education setting, where students spend more time working alone, have only telephone contact with tutors, and often have no contact with fellow students. We wish to alleviate these problems in the context of a distance education masters level course in Prolog. The difficulties students have in learning Prolog have been well documented (Fung, Brayshaw, du Boulay and Elsom-Cook, 1990; Taylor, 1988). It has been widely claimed that a crucial problem is students' inability to conceptualise the complex execution model (Pain and Bundy, 1987). This has lead to a great number of Software Visualizations (SVs) being developed, aiming to present a clear story of program execution. Software Visualization is the process of using techniques such as typography, graphic design, animation and cinematography to provide representations of a program and its execution. For an overview of SV research see Price, Small and Baecker (1991) and Stasko, Domingue, Brown and Price (in press). Each Software Visualization developed tends to be based on particular claims as to what kind of representation a student needs in order to learn effectively. Examples include the Prolog Trace Package (Eisenstadt, 1984; Eisenstadt, 1985) and the Textual Tree Tracer (Taylor, du Boulay and Patel, 1991), which aimed to show Prolog execution in fine-detail, using a representation close to the underlying source code. Conversely, the Transparent Prolog Machine (TPM) (Eisenstadt and Brayshaw, 1988; Eisenstadt and Brayshaw, 1991) used a graphical notation, allowing students to appreciate high level patterns indicative of various kinds of execution events without having to plough through a long textual history.

TPM was incorporated into a Masters level course on Prolog programming (Eisenstadt and Brayshaw, 1987), the notation being used across a range of media: textbooks, videos and software. The course is divided into three parts. The first part deals with the essentials of how Prolog works and how this is represented in TPM, and leads up to the point of understanding recursion. The second part of the course considers simple Artificial Intelligence programs, which involves the introduction of list processing and complex features of the language such as the "cut" and "not". The final part of the course explains the development of an expert system shell. Each part of the course is supported by exercises and video explanations using the TPM notation.

Although the developers acknowledged that the graphical notation of TPM would take longer to learn than some of the simpler textual SVs, they believed that this extra effort would be

rewarded as TPM would serve them through to full expertise, providing a "cradle to grave" story of Prolog, however recent empirical work and experiences teaching the course have highlighted four problems. Some of these are inherent to TPM, others are concerned with the educational effectiveness of SV in general or the added difficulties of teaching computer programming at a distance. Empirical work has shown that students can have problems mapping between the SV and the program. Experience teaching the course has illustrated challenges inherent in teaching computer programming at a distance. First there is the problem of engaging in discussions about complex programming concepts over the telephone or email. In such situations, it is difficult to establish the context of the students' problem, and their current level of understanding. In face-to-face discussions the tutor and student have a greater array of media at their disposal. For example they can use a whiteboard to illustrate their ideas, or can sit around the same terminal and watch a program work. Second, it is also difficult to keep in contact with, and support, students who tend to work at very different hours. Third, practically, it has become increasingly difficult to provide the range of software for ever more platforms and operating systems. We also have the constraint of using software that does not require expensive hardware. These issues led us to the development of the Internet Software Visualization Laboratory (ISVL), which provides a rich collaborative environment, allowing students to explore and receive tuition within a networked, platform independent environment. Although this paper describes using ISVL to teach Prolog, which has a particulalarly difficult execution model, our aim is not to encourage students to concentrate on the execution at the expense of other programming areas such as design and documentation. The most important feature of ISVL is that it allows student programmers to describe their program and how it works. This opens up new possibilities for stressing the importance of collaboration and communication skills within the computer science curriculum.

In this paper we first describe the current version of the Prolog course, then present empirical and anecdotal evidence as to how it could be improved, leading to a number of design goals which motivated ISVL. This is followed by a description of the ISVL environment and finally, a discussion of further work.

2. Context: an integrated programming course

The Masters course in Prolog was developed to be taught at a distance. It used the notation of the Transparent Prolog Machine (TPM) throughout textbooks, videos and accompanying software. A central aim behind using TPM, was that the notation would be equally applicable in both teaching purposes and as a debugging tool for novices and experts. This was based on the idea that a critical goal for designers should be to create software that was equally applicable for both novices and experts. Many, including Rajan (1986) have lamented the fact that programmers are often required to use different tools as novices and experts. As a novice they

use "student" tools which are easy to use but lack power and give a simplified perspective which has to be unlearnt. At some stage in the development of their expertise the programmer must leave behind "student" tools they have outgrown and move onto complex but powerful "expert" tools. The designers of TPM aimed to overcome this by providing an environment that provided a single truth about programming applicable to all levels of expertise. Eisenstadt and Brayshaw (1987) claimed that providing a rich and truthful account of the language confers a "cradle to grave" property upon the notation. They believe the notation and its implementation within TPM are equally suitable to the educational demands of the novice and the desire of experts for a powerful debugging tool. If correct, this means that the novice will not outgrow TPM as their knowledge of the language increases. As a result, the TPM model of execution was used to underpin an intensive Prolog course (Eisenstadt and Brayshaw, 1987). The aim of the course was to provide a clear, consistent and integrated model of execution which would continue to be beneficial as the level of expertise increased. By clear and consistent, Eisenstadt and Brayshaw meant that a complete procedural model of the language should be presented to the students from the first stage. Essentially, the aim was to tell the truth about the language throughout, regardless of the complexity of the constructs involved. The integration of the TPM notation allowed the student to integrate the debugging tool with the supplementary video and written materials. Each component of the teaching material is built around the TPM notation. Even the workbooks provided for self assessment contained examples where students are required to display their knowledge of Prolog execution by completing TPM-style descriptions of the program.

```
parent(pam, angela).
parent(angela, ann).
grandparent(X, Y) :- parent(X, Z), parent(Z, Y).
```

Figure 1. *The Prolog code for the grandparent program.*

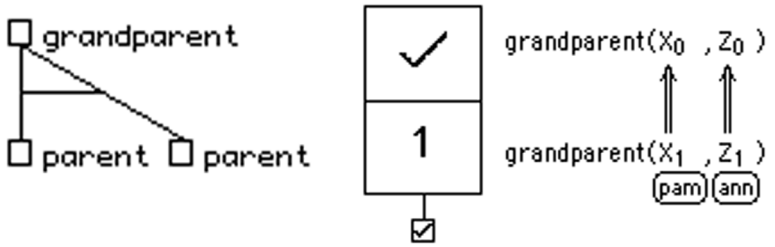

Figure 2. *A TPM visualization of the above grandparent program. The Coarse-Grained View of the whole execution is shown on the left whilst a Fine-Grained View of the top node is shown on the right.*

TPM uses an AND/OR Tree model of Prolog, where execution is shown as a depth first search. TPM incorporates two levels of granularity. The coarse-grained view uses an AND/OR tree diagram to provide an overview of how clauses are interrelated during execution. Fine-grained views giving the unification history for a particular node are obtained by clicking on the node in question. The fine-grained view uses a separate notation to show variable binding. This makes explicit the cross variable dependencies between subgoals. Figure 2 shows a coarse grained (left) and fine-grained view of the grandparent program (Figure 1).

3. Design issues

We have taught Prolog at a distance using the integrated course for eight years. During this time a number of problems have been identified, either anecdotally or by empirical analysis of how students are able to use TPM during the early stages of learning. The first section will report empirical evidence as to the educational effectiveness of TPM. Some observations will then be made concerning the potential role of SV in general as a tool for exploratory learning. Third, how having to teach at a distance impacts on the course will be considered. Fourth, pragmatic issues relating to how new technology can be incorporated into an existing course will be outlined. The issues raised will be used to motivate the design of ISVL.

3.1 Students using TPM

Our empirical work has looked at how students are able to use TPM in the early stages of learning Prolog. The students were studying another of our courses which involved some simple Artificial Intelligence programming in Prolog. Their expertise can be paralleled to that of students in their first few weeks of the masters course. We describe the results considering four aspects: the students' overall performance on the given task, and three measures taken from a protocol analysis of them performing the task. Protocol analysis was used to give a fine-grained account of how the students were able to use TPM, and the kinds of difficulties they had. A reliance on overall performance measures alone could belie important cognitive evidence of how the students used the SV.

The three types of cognitive evidence investigated were: information access from the display, strategies adopted to aid comprehension, and misunderstandings of the SV or execution. A review of the literature identified these three forms of evidence as being central to program comprehension. As a main purpose of SV is to aid program comprehension (whether within an educational context for novices or as debugging aids) these could be used to assess the effectiveness of each SV. A full account of the methodology and empirical work can be found elsewhere (Mulholland, 1995; Mulholland, in press).

The study was carried out involving 64 Open University (OU) residential school cognitive psychology students taking the Artificial Intelligence project. Students taking the project are required to model a simple cognitive model in Prolog. Each residential school project lasts approximately 2.5 days.

Initially, a pre-test was administered to ensure the students understood the basics of control flow and unification before commencing the experiment. Questions covered the order in which goals and subgoals are tried and the Prolog unification algorithm. The subjects were given five minutes to familiarise themselves with a program presented on a printed sheet. They each retained a copy of this program throughout the experiment. They were then asked to work through the visualizations of four versions of the program which had been modified in some way. Their task was to identify the difference between the program on the sheet and the one being visualized. They had no access to the source code of the modified versions. After five minutes the subjects were given the option to move onto the next problem. Verbal protocols were taken throughout. Subjects worked in pairs during the experiment to facilitate a natur listic protocol.

Subjects using TPM performed poorly on the task. The protocols helped explain why TPM subjects accomplished far less than those using a textual trace. TPM subjects made fewer control flow and data flow statements though did comment far more on the high level goals of the program. This suggests that although the subjects could gain a vague understanding of the program on a higher level of abstraction they had difficulty using the trace to extract more fine grained information. TPM subjects primarily used a mapping strategy to map between the SV and the source code. Unfortunately, this strategy was often carried out ineffectively, subjects being confused by the lack of affinity between the tree representation and their textual program. TPM subjects also attempted to get an overview of what the SV was showing them. This particular strategy (called the overview strategy) was not found with subjects using any of the other SVs evaluated. The overview strategy was used to gain a more global account of the execution and can be interpreted as the early development of visual Gestalt strategies (Petre and Green, 1993), where users derive information implicitly held within a graphical display, only revealed by patterns occurring within it. As the subjects had little Prolog expertise and only a short exposure to the SV they were not able to develop the general overview approach into a more well defined set of strategies (see review discussion on teaching debugging methodology to novice programmers). Subjects could use the SV to spot basic changes such as incorrect relation names or an incorrect ordering of subgoals but were unable to derive more complex implicit information.

TPM subjects also frequently lost their position in the execution. It could therefore be that using a single Gestalt throughout the execution could be a false blessing when novices are using the

trace unaided. TPM subjects also found it difficult to deal with the large number of windows that can result from zooming into a number of nodes within the tree. Although the TPM subjects fared less well, they still reported having found it to be a fairly useful teaching aid. The results indicate that students in their early stages of the masters course probably do not seamlessly "grow into" the notation in the way envisaged by the designers. *The implementation of TPM within the ISVL environment should therefore support students in: (i) working out how to map effectively between the tree representation and their textual code, (ii) understanding the temporal position within the execution and its historical context, and (iii) navigating the multiple views within the SV effectively.*

3.2 Software Visualization as an educational technology

A major aim of SV technology is to support computer science education. SV is the second large attempt to employ software to support computer programming education, the first being Intelligent Tutoring Systems (ITSs). The main difference between ITS and SV is that ITSs require the system to build a model of the student in order that the most effective advice can be given. SV on the other hand essentially presents the program information according to some notational formalism without gaining an internal model of the student. A number of attempts have been made to provide intelligent programming tools to support both debugging and computer science education, including the Programmer's Apprentice (Waters, 1985) and the Lisp Tutor (Anderson and Reiser, 1985). Eisenstadt, Price and Domingue (1992) identify a number of problems with the ITS approach. The most fundamental is that ITSs require very large bug catalogues in order to build the required models, making it very difficult for ITS to scale beyond even the simplest programs.

This kind of issue lead to a shift in focus toward SV as a more promising solution. SV aims to provide a clear story of the language which the student can explore and learn from. There is therefore no need to develop bug catalogues or consider what kind of response should be given to the student when a bug is located, the central issue for SV being the clarity of the external representation used to present the language and its execution.

Detailed evaluation suggests SV has its own difficulties (Mulholland, 1995; Mulholland, in press; Mulholland and Eisenstadt, in press). One serious problem is the students' tendency to reinterpret the display to fit their own expectations. For example, often when students had a misunderstanding of the program execution, the display would be interpreted in such a way as to be consistent with their false expectations, rather than the display leading the students to challenge and reformulate their understanding. This suggests that SV cannot be relied upon as a stand-alone educational tool and must be carefully incorporated into, and supported by other aspects of the educational environment.

Other studies have illustrated that the educational benefits of SV are not as clear cut, or easy to establish as many thought. Stasko, Badre and Lewis (1993) considered the educational benefits of SV in their study of a visualization of a priority queue algorithm. Half of the students were provided with the visualization and the program, while the other half were just given the program. The SV was found to only slightly assist student comprehension. They suggested one of the reasons the SV may have been of little help was that students were not aware of how their knowledge was to be tested prior to viewing the visualization. The students therefore had no clear goal to pursue during the comprehension phase of the experiment. They suggest a clear task motivation may have helped the students to gain far more from the representation than they actually did. This interpretation was borne out by a further study by Lawrence, Badre and Stasko (1994). They found that students who were able to act interactively with the algorithm as well as view the algorithm in a classroom setting had a deeper understanding. Similarly, Byrne, Catrambone and Stasko (1996) found that students who were encouraged to predict how the algorithm would develop rather than just passively viewing it developed a greater understanding.

Overall, these findings suggest that the extent to which the student is actively involved when using the SV affects their learning outcomes, though even when students are actively involved, they can interpret the SV incorrectly, rather than learning from it. Students therefore should be encouraged to use the SV interactively to meet some set objective, though they should be monitored or scaffolded when learning by exploration to keep them on track. *ISVL should therefore both allow students to interact with the SV and also be usable in collaborative ways with the tutor to counter circumstances when personal exploration can run into difficulties.*

3.3 Teaching computer programming at a distance

Teaching any course at a distance involves two kinds of staff: the course team and the tutors. The course team are responsible for developing the course. The tutors work on the "front line", teaching the course developed by the course team to the students. The tutors are the ones who communicate directly with the students, as well as marking and providing feedback on assignments. On very small courses, members of the course team may undertake a significant amount of the tutoring, though this is rarely the case. Many courses have thousands of students enrolled at any one time. These larger courses may provide opportunities for face-to-face contact between tutors and students at tutorials and residential schools. This is not feasible on more specialised courses, having a smaller number of students spread over a large area. This is the case for our course in Prolog programming, whereby communication between tutor and students is restricted to telephone and email contact. Tutors have often commented how difficult it is to conduct technical discussions using these impoverished media. For example, tutors will often have to deal with a student's programming problem over the telephone. A common telephone

conversation between a student and tutor runs something like this:

student: "I'm having problems."
tutor: "What sort of problems?"
student: "I don't understand."
tutor: "What don't you understand?"
student: "I don't understand anything about this latest assignment."
tutor: "Is there anything in particular about the assignment that you don't understand?"
student: "It doesn't make sense"
tutor: "Can you outline any concepts which are causing your problems?"
student: "Recursion I think"
tutor: "Anything in particular about it?
student: "I just can't get it to work"
tutor: "What have you typed in?"
student: "Well the first line is...."

The reason for the poor interaction above is because it is very difficult for the tutor in such circumstances to understand what precisely the student is having a problem understanding. This means either the tutor has to guess and plough through areas of the course (which could confuse the student further) or require the student to undertake the difficult task of explaining what it is that they do not know. There is of course the added difficulty of describing a program over the telephone. An earlier attempt to teach students at a distance was the virtual summer school (Eisenstadt, Brayshaw, Hasemer and Issroff, 1996). Use was made of technology such as video conferencing to provide a richer level of contact between students and tutors (see review comments on what richnes of media means in the context of this paper). *An important design aim of ISVL is that it will provide an internet communication environment within the course, used synchronously to establish a context within which the tutor can understand and help the student with their difficulty.*

Providing a rich synchronous communication medium will not however solve all of the problems inherent in teaching at a distance. Students study part-time and have to fit their course work around other work and personal commitments. In these cases, email is currently relied upon as the sole form of communication. Unfortunately though, like the telephone, email is an impoverished medium in which to communicate programming concepts. This is particularly the case when the student is best able to communicate their problem most effectively using the graphical TPM notation which is used throughout the course; which is not particularly amenable to the ASCII character set. *It is therefore important that ISVL allows students and tutors to work asynchronously, enabling students easily to leave meaningful queries or thoughts for the tutor or other students to view at later times.*

Within a conventional university, practical sessions within a programming course tend to occur in departmental laboratories, where all students are using a uniform platform. Within a distance education setting, this is difficult to establish, as students will have a range of hardware, or possibly different operating systems running on similar hardware. Making a particular hardware and software configuration mandatory will vastly reduce the number of available students, though supporting a range of platforms, particularly when complex software is being used, is difficult. Additionally, in order to keep costs down we can only assume that students have a run-of-the-mill computer and standard modem which can not transmit high bandwidth data such as video. Also students will be connecting via a modem from home, meaning bandwidth must be efficiently used. *ISVL must therefore use the internet in way which is both platform independent and makes efficient use of the bandwidth resources.*

3.4 Incorporating new technology into an existing course

Distance education courses, even when taught to some extent over the internet have a definite life cycle and cannot be transformed over-night. This is not only true when the course as a whole is changed. It also applies even when one component is altered. This is because alterations to one component of the educational materials can have serious knock-on effects, due to the way the materials are integrated and cross referenced. As was mentioned earlier, the Prolog course uses the TPM notation throughout. Although problems have been noted with TPM, providing a extra materials using a completely different notation could do more harm than good. Even if the new notation has been found to be a very effective educational tool, foisting more notations to be learnt on the students could just cause confusion and detract from the aim, which is to teach Prolog.

This illustrates some of the pros and cons of having a tightly integrated course. A high level of integration has the advantage of consistency: using the same notational "story" of the language throughout. This allows students to benefit from this familiarity and appreciate links between components of the course. Skills on how to read and interpret the notation on one part of the course will be useful when learning other components. The drawback of a tightly integrated course is that there is less room to manoeuvre when the course team wishes to modify components in isolation or add extra modules to the course. New materials have to be consistent with the rest, unless the entire course is to be rewritten.

As part of the perceived difficulty with the current course is the notation itself, it may appear better to just change the entire course at once. This itself would be both problematic and risky. Rewriting an entire course from scratch takes a great deal of resources. Typically an OU course involves between 5-20 personnel for 2-5 years. There would be the additional costs of retraining

the staff. As there are identified problems with the course this could be a plausible route, though as yet it is not clear what notation or educational approach could be guaranteed to work better. It would be foolish to undertake such a wholesale change without a greater degree of evidence. All of the detailed empirical work focuses on the early stages of students learning Prolog. Once some initial learning curve has been transcended, the benefits of the notation may come to far outweigh any initial climatisation problems. A more sensible approach would be to use the greater level of communication permitted by ISVL to support the existing course, simultaneously carrying out a detailed long-term investigation of how the course currently fares. *For this reason, ISVL will be integrated with the existing materials, providing a laboratory for course development, allowing evaluation studies to be conducted more easily.*

3.5 Summary

The design issues which motivated ISVL are summarised below:

i) Modifications to TPM which will help students map between TPM and the code, appreciate the temporal development of the SV, and deal effectively with multiple views;
ii) Maintain a sufficient degree of integration with existing course materials;
iii) Encourage active and collaborative use within the new environment, rather than passive stares;
iv) Support synchronous communication for one to one help;
v) Support asynchronous communication for helping those working at different times;
vi) Be platform independent whilst making efficient use of bandwidth resources.

The next section will first describe the design of ISVL in terms of its functionality and interface, dealing with issues (i) through (v). A range of scenarios will then be presented illustrating how the environment can be used, illustrating issues (iii) through (v). Following this, a description of the implementation is given, explaining how ISVL is realised in a way which deals with issue (vi).

4. Approach: The Internet Software Visualization Laboratory (ISVL)

This section explains how the identified design issues were realised within ISVL. Firstly, an overview of the design decisions is given. This is followed by a number of scenarios showing how tutors and students can be expected to work with ISVL. Finally, the underlying architecture is described.

4.1 Design overview

ISVL aims to use internet technology to provide an effective educational resource which complements and builds on standard course materials. The design of ISVL has a number of core features, dealing with the issues raised in the previous section. ISVL, on the whole, adopts the TPM notation found in the other course materials. A number of changes though have been made to scaffold the students use of TPM. Changes to the notation facilitate the process of mapping between TPM and the code, and the appreciation of the temporal development of the SV. Second, the new interface deals with the problems students had in dealing with the multiple views.

The full version of TPM provides two views of program execution: coarse-grained and fine-grained. The coarse-grained view shows the overall hierarchy of goals making up the execution. The fine-grained view gives details of variable bindings for a selected goal within the coarse-grained hierarchy. Within the SV representation, the conventional fine-grained view is removed and replaced with a parallel textual account of the execution, giving the same information. This mean that the notation will still be integrated with the rest of the course, but will have some distinct advantages. Firstly, the textual, historical account will help the students to appreciate the temporal context for the execution events being shown. This is very important. Without a firm idea as to how the current execution "clock tick" fits in, it is not possible for the student to make strategic use of the information they access. As was discussed earlier, the ability of students to both review presented information and test and make predications as to how that information develops, is strongly related to their ability to successfully comprehend the program and how it works. Secondly, the change will mean that students will not have to zoom-in to particular nodes to acquire fine-grained information. The evaluation of TPM found that students had particular difficulties with the fine-grained views, both cognitively, in terms of finding out how they fit in with the rest of the display, and also in terms of the navigational demands of dealing with numerous windows.

The parallel textual trace selected is based closely on the Prolog Trace Package (PTP) (Eisenstadt, 1984) (Figure 3) which was found in an earlier study to be a highly effective education tool for students in the early stages of learning Prolog. Students were found to able to make effective use of PTP after only a brief demonstration. The close affinity of PTP to the underlying code means that the students are able to map effectively between the display and program being visualized, and able to easily develop strategies to explore the control flow and data flow information being shown. It will therefore provide the students with the choice of simple intermediate representation of the execution, to work as a safety net during their initial attempts at using TPM. Their use of ISVL will be further supported by code highlighting to aid the process of mapping between the representation and program. This will establish an effective trade-off between

providing the most effective environment, and providing an environment consistent with the other course materials.

```
1: ? grandparent(_1, _2)
2: > grandparent(_1, _2) [1]
3: ? parent(_1, _3)
4: +*parent(tom, liz) [1]
5: ? parent(liz, _2)
6: -~parent(liz, _2)
7: ^ parent(tom, liz)
8: < parent(tom, liz) [1]
9: +*parent(pam, bob) [2]
10: ? parent(bob, _2)
11: +*parent(bob, ann) [3]
12: + grandparent(pam, ann) [1]
```

Figure 3. *The Prolog Trace Package (PTP), showing the execution of the grandparent program. The symbols above denote the following: '?' a goal, '>' unification with the head of a clause, '+' a goal succeeding, '-' a goal failing, '<' backtracking out of a goal and '^' redoing a goal.*

As well as being used by students for their personal exploration of programs, it is envisaged that ISVL will have an important role as a communication environment between students and tutors. It is important that ISVL should support both synchronous and asynchronous communication. Synchronous communication using ISVL will complement the current position taken by telephone contact, providing immediate synchronous help to one or more students on specific problems. This has similarities with the work of Brown and Najork (1996) in their development of Collaborative Active Textbooks (CAT). Like ISVL, CAT is a web-based SV environment, though CAT is restricted to animations of specific computer science algorithms. The CAT environment allows the same animation to be run simultaneously to a number of machines, and is intended for use in a synchronous, classroom style setting. The view and speed of the animation can be controlled remotely by the tutor. This form of synchronous demonstration of programs is possible within ISVL, though within ISVL the animations are not canned, being created on the fly, from the program submitted by the tutor or student. ISVL also permits asynchronous help, whereby tutor demonstrations can be archived for the students to access later. This will build up an extra resource of teaching materials in response to student demands, and is particularly important, because within a distance education setting, students tend to work at different times and a classroom style approach is only occasionally appropriate.

4.2. Scenario

In this section we shall describe how ISVL embodies the approach outlined in the introduction, using a scenario involving a hypothetical student Bill and tutor Ingrid. Within the scenario we shall show how ISVL supports students debugging their programs and how ISVL provides a rich communication medium between students and their tutor. In order that the reader can fully appreciate the scenario, all communications are shown as archived ISVL movies, though any of the communications shown could have been carried out synchronously.

Before presenting the scenario, we will briefly introduce the ISVL interface. Figure 4 shows the ISVL Prolog client running on a Netscape™ Web browser. The client contains a visualization of a Prolog program. This was obtained in the following way. First the query 'qsort([4,3,1], _y)' (find a -y such that y is the sorted version of [4,3,1]) was typed into the Prolog Query window (1). Then the "Evaluate" was button pressed. The result of the query "NO" (indicating that the query has failed), the value of variable _y, and a TPM coarse grained view (2) were then returned. A fine-grained view of the arrowed node was obtained by clicking on the node (3). The user can step through the execution using the video recorder style buttons in the control panel. The user can play through the execution using the  button. The speed is controlled by the scroll bar in the control panel.

The scenario is divided into six parts[1]:

1. Bill has a bug
2. Bill spots the bug
3. Bill fixes the bug
4. Bill needs help
5. Ingrid to the rescue
6. Bill shows off

[1] *In the Web version of this article, each part contains an interactive ISVL movie as a Java applet.*

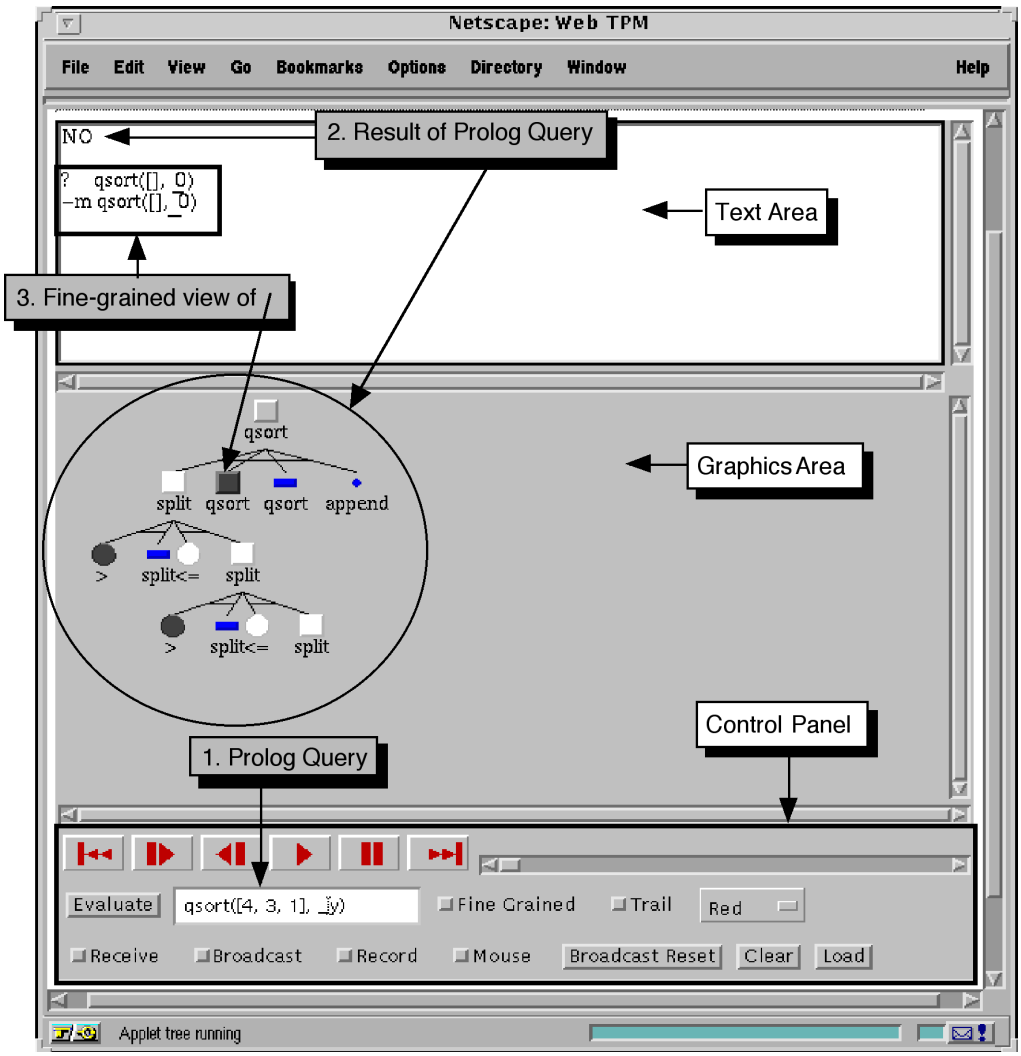

Figure 4. *The ISVL interface.*

4.2.1. Bill has a bug

Bill is working on an exercise from his course workbook. The exercise asks students to write a sorting program, called qsort, which uses the quick sort algorithm. The quick sort algorithm works by splitting a list around an element into a list of lower numbers and a list of higher numbers which are then recursively sorted. The program should take an unsorted list and return a sorted list. For example, the query:

```
qsort([4, 3, 1], _y)
```

where _y represents an unbound variable, should return:

```
YES _y=[1, 3, 4]
```

signifying that the query has been proved and the value of _y is [1, 3, 4]. Bill's solution is shown below.

```
qsort([X|Xs], Ans) :-
      split(Xs, X, LOs, HIs),
      qsort(HIs, SortedHIs),
      qsort(LOs, SortedLOs),
      append(SortedLOs, [X| SortedHIs] Ans).
split([], X, [], []).
split([X|Xs], Crit, LOs, [X|HIs]) :-
      X > Crit,
      split(Xs, Crit, HIs, LOs]).
split([X|Xs], Crit, [X |LOs], HIs) :-
      X =< Crit,
      split(Xs, Crit, LOs, HIs).
```

Bill loads his solution by copying it into the text area and clicking on the "Load" button in the control panel.

Bill types his query:

```
qsort([4, 3, 1], _y)
```

and clicks on the "Evaluate" button. A TPM visualization is returned. Bill can now step through the execution using the button in the control panel, looking for the bug...

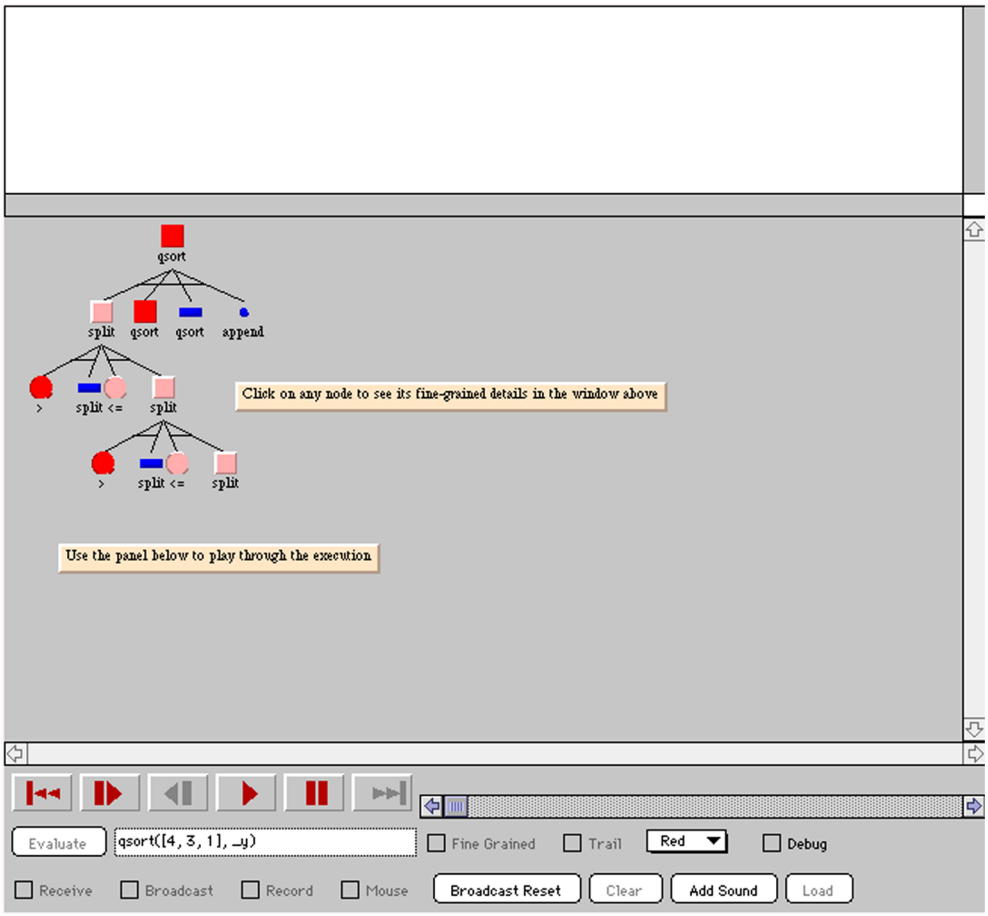

Figure 5. *Bill has a bug.*

4.2.2. Bill spots the bug

When stepping through the execution using the button in the control panel, Bill spots the bug. He notices the suspicious failure of the first qsort node and investigates this further by clicking on the node to obtain a PTP style fine-grained view. The fine-grained view tells Bill that there is no match when the list to be sorted is empty. Bill makes an ISVL movie of his bug spotting activity...

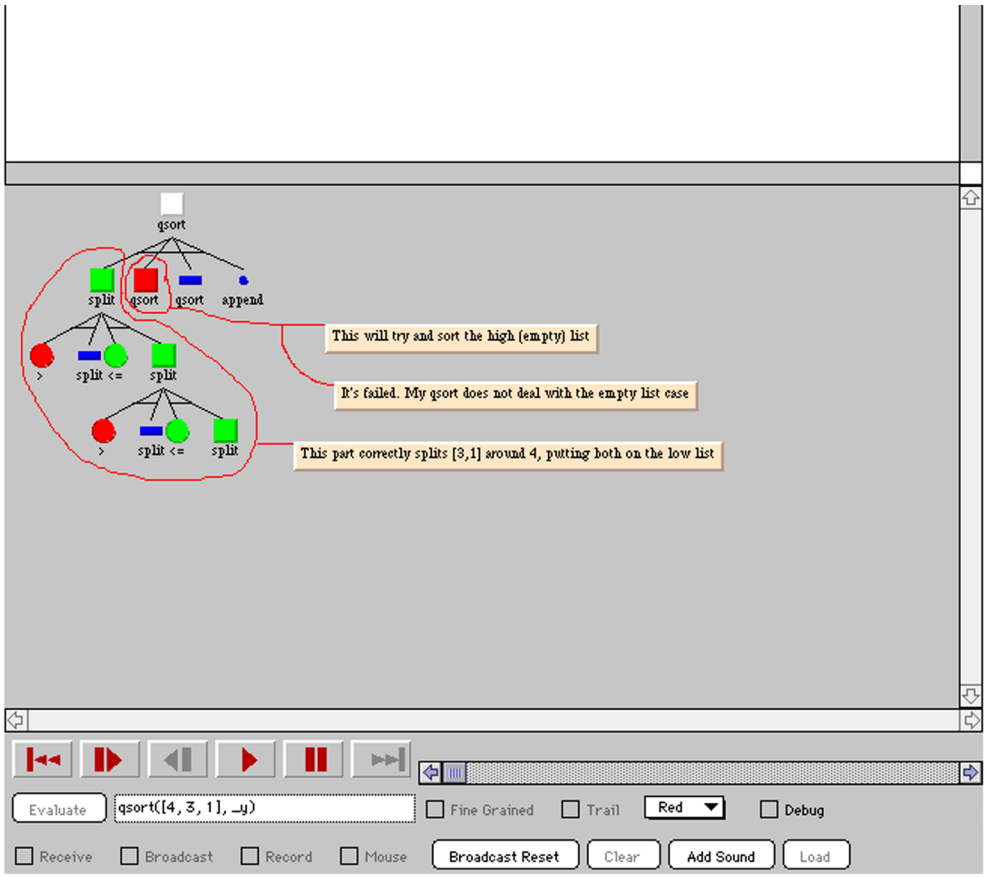

Figure 6. *Bill spots the bug.*

4.2.3. Bill fixes the bug

Bill now modifies his program to deal with the empty list case. His new program is shown below.

```
qsort([], []).    %This clause deals with the empty list case
qsort([X|Xs], Ans) :-
      split(Xs, X, LOs, HIs),
      qsort(HIs, SortedHIs),
      qsort(LOs, SortedLOs),
```

```
        append(SortedLOs, [X| SortedHIs] Ans).
split([], X, [], []).
split([X|Xs], Crit, LOs, [X|HIs]) :-
        X > Crit,
        split(Xs, Crit, HIs, LOs]).
split([X|Xs], Crit, [X |LOs], HIs) :-
        X =< Crit,
        split(Xs, Crit, LOs, HIs).
```

Bill makes a movie to illustrate his progress.

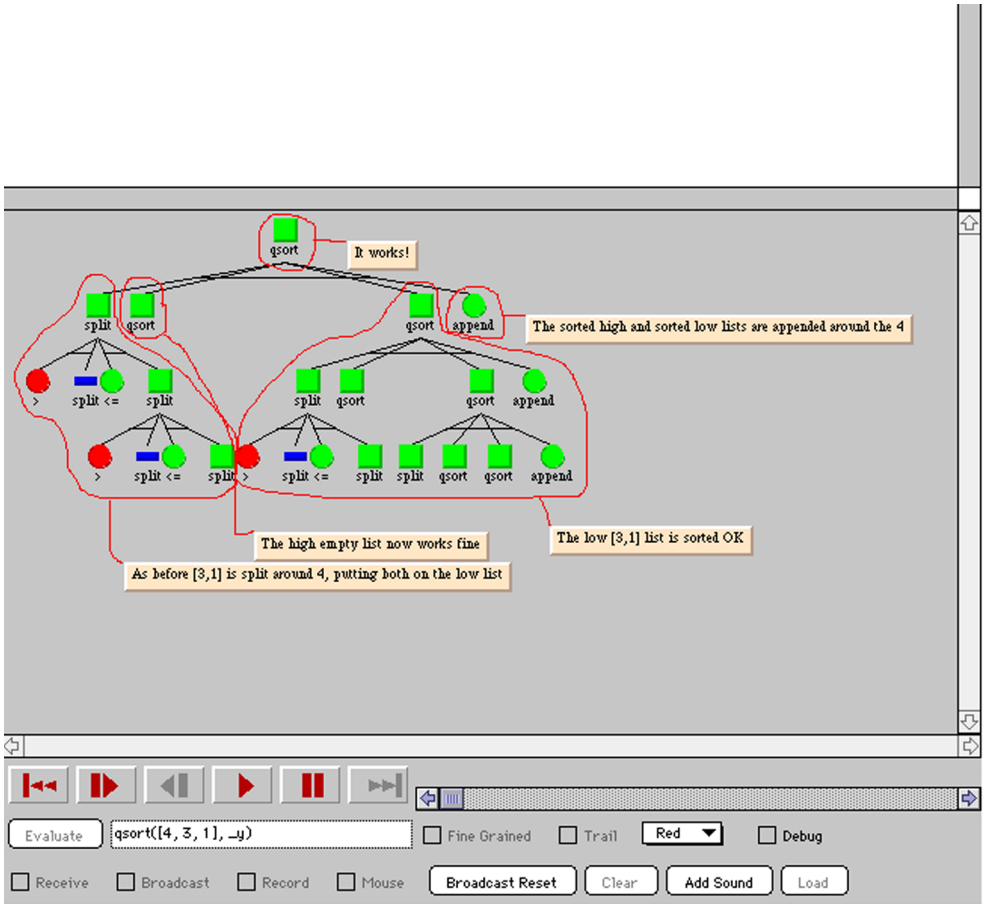

Figure 7. *Bill fixes the bug.*

4.2.4. *Bill needs help*

Some time later Bill tries his code with the query:

```
qsort([4, 2, 3, 1], _y)
```

and gets the result:

```
YES _y=[2, 1, 3, 4]
```

Even with the visualization Bill fails to track down the bug, so he makes a movie asking his tutor Ingrid for help...

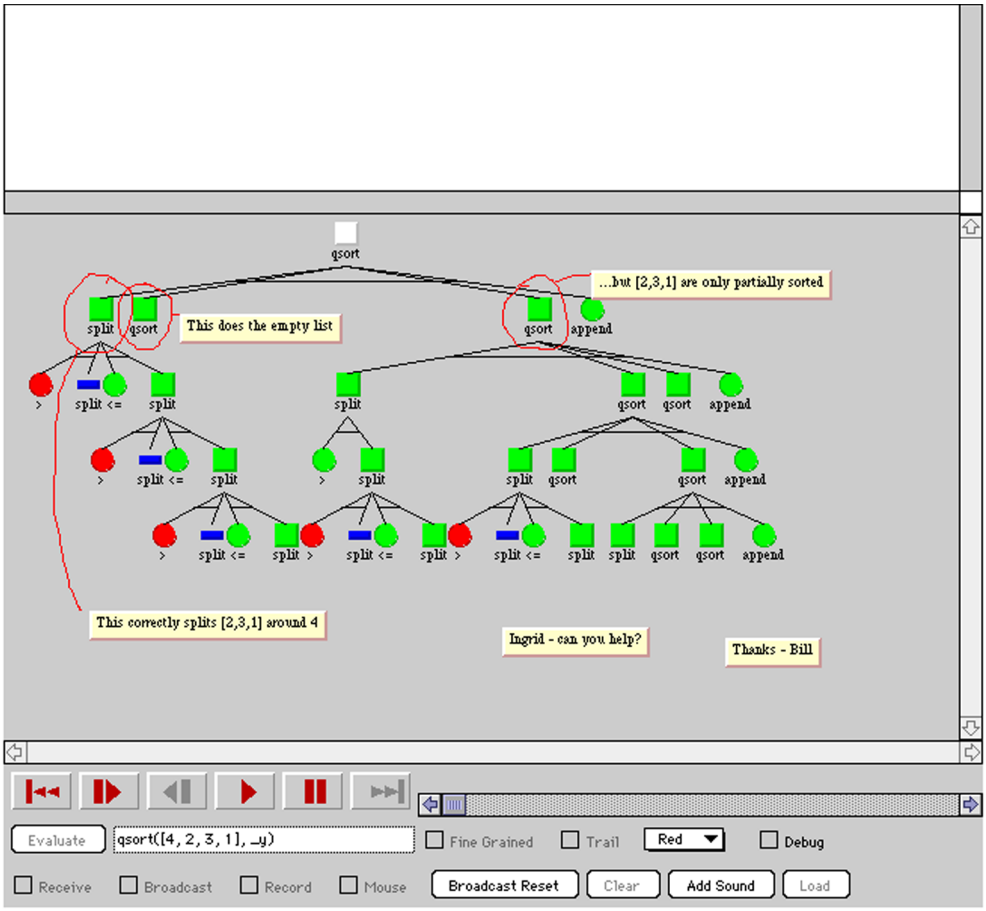

Figure 8. *Bill needs help.*

4.2.5. *Ingrid to the rescue*

Later that day, Ingrid views the movie Bill has made. By viewing the tree and clicking on the nodes to get a fine-grained view she is able to spot the bug. The split part of Bill's program is working incorrectly. Splitting the list [3, 1] around 2 should yield the list [1] (numbers less than 2) and the list [3] (numbers greater than 2). Ingrid makes an annotated movie to direct Bill's attention to the buggy part of the code...

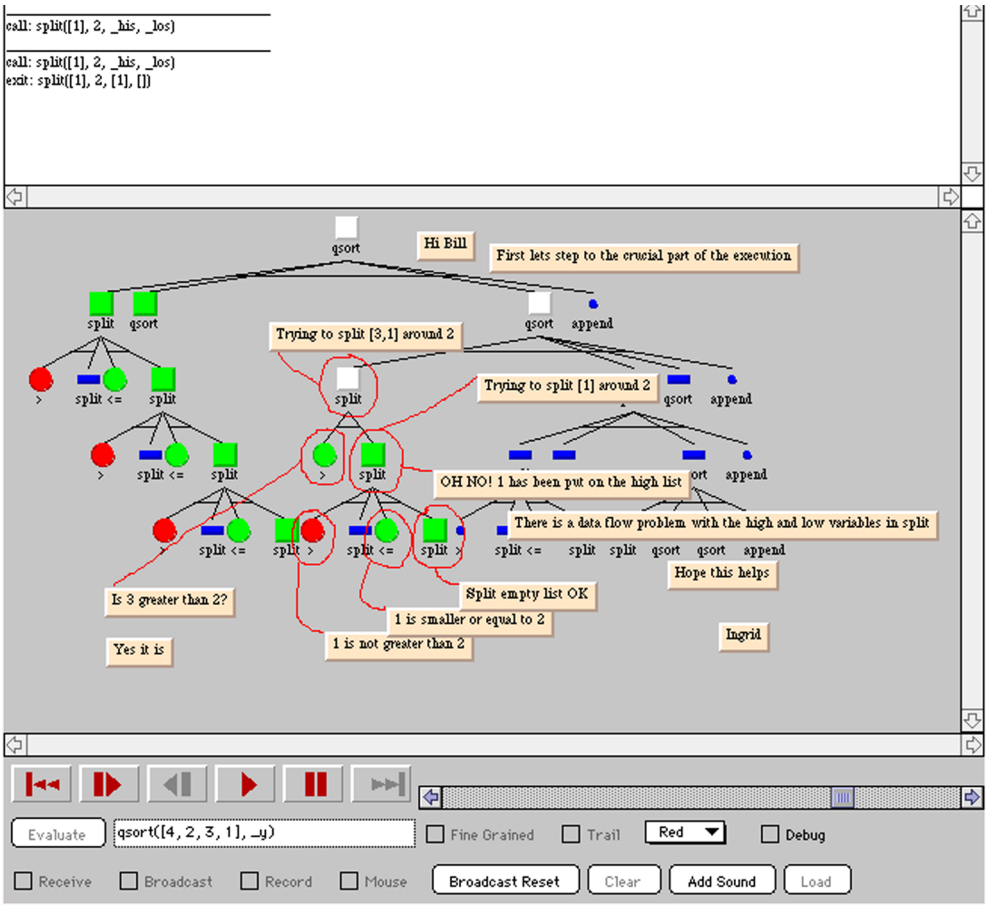

Figure 9. *Ingrid to the rescue.*

4.2.6. Bill shows off

Having viewed Ingrid's movie, Bill is able to fix the data flow error in the split part of his program. His new program is shown below:

```
qsort([], []).
qsort([X|Xs], Ans) :-
      split(Xs, X, LOs, HIs),
      qsort(HIs, SortedHIs),
      qsort(LOs, SortedLOs),
      append(SortedLOs, [X| SortedHIs] Ans).
split([], X, [], []).
split([X|Xs], Crit, LOs, [X|HIs]) :-
      X > Crit,
      split(Xs, Crit, LOs, HIs]).   %Low and high swapped to
                                     fix data flow bug
split([X|Xs], Crit, [X |LOs], HIs) :-
      X =< Crit,
      split(Xs, Crit, LOs, HIs).
```

Bill is so proud of his efforts that he makes a movie to show off his working qsort program....

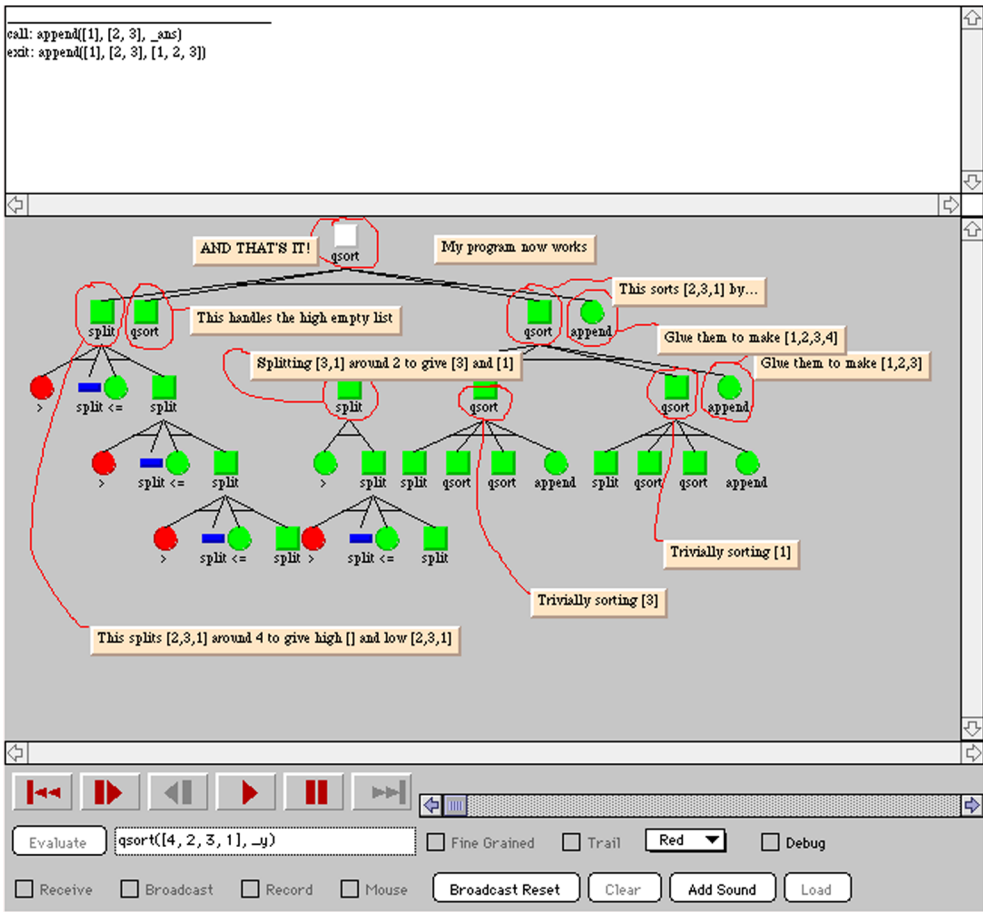

Figure 10. *Bill shows off.*

4.3. ISVL architecture

The fact the ISVL is composed of a central server and a client requires that we think about where the components should go. In general data which is on the client and is therefore local will be rapidly accessible. But local data increases start up time (because the Java code has to be downloaded). Storing structures on the server reduces the student requirements for processor power and memory and facilitates collaboration. This has to be balanced, however, with the fact that access to server structures are slower and have a 'dial-up' cost.

ISVL is based on our framework for creating SVs called Viz (Domingue, Price and Eisenstadt., 1992). We have split Viz between the server and client as shown in Figure 11. The program execution history and views are stored on the server. The history is made up of events which cause players to change state (see Domingue et al., 1992 for details). A view consists of a series of locations (e.g. the TPM SV uses a tree based view). A mapping maps a history structure to an icon/image class (e.g. a Prolog goal is mapped onto a coloured square). The views and maps form the basis of the ISVL HTTP protocol which is used to transmit the SV. The client interprets the view and map data to produce the on screen images which the user manipulates through a navigator. The navigator allows large execution spaces to be viewed by using techniques such as scale and compression.

Using a protocol specific to SV systems means that the amount of data transferred is minimal. Reproducing views and maps on the client means that an SV can be examined without the need for repeated accesses to the server.

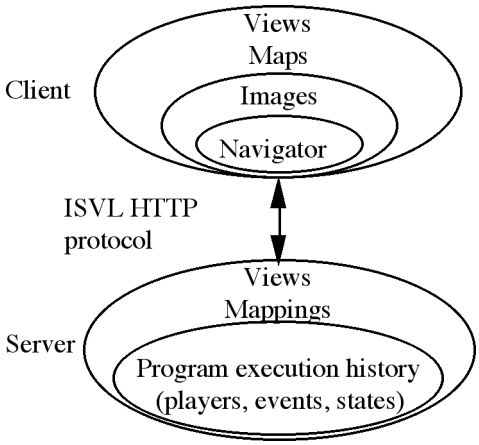

Figure 11. *The server/client split in ISVL.*

On the client side the mapping module converts the ISVL HTTP stream into textual and graphical representations. These representations are then transformed and presented on the screen by the navigator. The user interacts with the visualization using the navigator. The navigator controls panning, zooming, local compression and expansion, and moving forward and backward in time through the program execution space. User requests for non-local information, such as visualizing a new execution result in requests sent to the ISVL HTTP Processor.

Users can elect to broadcast or record their interface actions. In this event their actions are sent

via the HTTP Processor to the broadcaster or storage modules. The broadcaster simply sends whatever it receives to the currently receiving clients. The client-server architecture of ISVL is similar to that of Baker, Cruz, Liotta and Tamassia (1996) whereby the bulk of the work is done on the server, the interface being created by the Java client.

5. Discussion

ISVL aims to provide a richer environment for teaching Prolog over the internet, making improvements on the existing course whilst still being consistent with the rest of the materials used by the students. Changes to the notation will, we believe, scaffold the students' use of the TPM notation. The richer level of communication permitted will add an extra dimension often difficult to achieve in a course taught at a distance. The environment will also allow us to investigate how students are able to use the representations over time, which will permit larger changes to the course to be more informed. The architecture of ISVL will permit its application to other programming language courses in the future. Our planned work on ISVL can be divided in three main threads: technological, theoretical and social.

The technological....

From a technological viewpoint, we are working to extend the use of sound in synchronous and asynchronous communication sessions. Work is currently in under way within the KMi Stadium project (Eisenstadt, Buckingham Shum and Freeman, 1996) to develop an audio delivery tool in Java which can be easily synchronised with screen activities. This would provide an easy to use audio-visual movie development environment, which hopefully would not only be used by tutors to deliver asynchronous help, but would also be effectively used as a method whereby students could share ideas. The current version of ISVL allows the inclusion of voice as well as textual annotations, though these make inefficient use of bandwidth and are difficult to synchronise with screen events.

Secondly, we are developing a movie searcher, to help students select archived movies appropriate to the particular programming concepts they are working with at any particular time. This is based on the search agents incorporated within the Multiple Representation Environment (MRE) (Brayshaw, 1994). Within MRE, users are able to select an execution pattern directly from the display. The selected pattern can then be identified in other programs. Within ISVL, student will be able to highlight a particular execution pattern and receive a list of movies within which the pattern features. This will ensure that students can easily select appropriate resources from the ISVL archive.

The theoretical....

From an theoretical point of view, future work will focus on the role of representations, such as TPM, in learning computer programming. A central ethos of the existing course is that a single representation should be used throughout all materials and for the full length of the course.

There may, however, be situations where sticking to a single representation may not be the most effective approach. Two particular issues will be investigated in the future: (i) using different representations appropriate to each stage of the learning process and (ii) simultaneously using a range of representations to encourage students to understand the same program in more than one way.

An interesting investigation would be to compare the learning outcomes using a single SV throughout the course (as in the current course) with using a simpler notation at the early stages and then shifting to the more complex TPM notation at a later stage. The empirical work presented earlier suggests that TPM may not be a very suitable SV for the early novice, though it could be argued that the difficulties confronting the early novice are possibly outweighed by the extra cost of having to transfer between different notations during the course, should TPM be presented at a later stage. An alternative view would be that novices will learn more effectively through a notation suited to their characteristics and strategic capabilities than through a notation which requires them to aspire to expert strategies from an early stage.

A further issue we wish to pursue is whether providing simultaneous multiple views of a program is educationally beneficial. This technique could be used to present the student with different perspectives on the same program. Karmiloff-Smith (1979; Karmiloff-Smith, 1992) provides some evidence that representational diversity is a crucial component in the attainment of expertise. Her theory of human development has highlighted the importance of what she termed Representational Redescription (RR). RR is the process of learning by iteratively rerepresenting knowledge in different formats. Karmiloff-Smith's observations of children have found that the child will initially develop competence at some skill. The focus of this stage is what she terms "behavioural mastery" rather than the development of a deeper level of understanding. The child will then "unpack" the structures underlying that competence in order to make them more explicit to the individual. This occurs by initially making the new internal (rather than the external) representation the main driving force. During this phase new kinds of errors previously absent appear as a result of the incompleteness of the internal representation. Later the external and internal representations become reconciled leading to a new competence underpinned by a deeper understanding. This process of unpacking knowledge to refine the internal representation is facilitated by iteratively redescribing the knowledge in different internal representational formats. Davies (1994) provides an interpretation of his own findings which suggests that such an effect may exist within the development of computer programming expertise. He found the recall of focal and nonfocal lines within Pascal programs to develop nonlinearly, which would be consistent with knowledge restructuring stages during expertise development within computer programming.

The usefulness of multiple representations though may not be confined to particular stages

within the learning curve, rather having a particular property amenable to relating particular kinds of concept. Petre, Blackwell and Green (in press) suggest that multiple representations could provide a "useful awkwardness", whereby the process of translating between representations forces a deeper level of understanding, uncovering issues that may otherwise have been missed. ISVL can be used to present a range of diverse perspectives on program execution. These could provide external support and prompting for taking a number of perspectives on the same program.

The social....

The restricted forms of communication offered within conventional distance education not only constrain the ways in which the tutor is able to communicate with students but also the extent to which fellow students are able to communicate. Within a conventional university, students have ample opportunity to discuss ideas, by virtue of congregating within the same department. Within a distance education setting this requires more effort on the part of the students, making collaboration and the sharing of resources more difficult. The introduction of technology such as ISVL, incorporating rich synchronous and asynchronous communication may transform this situation. These new forms of communication may well have great advantages for the senders as well as the recipients of educational resources. Research has considered the education benefits of a student explaining a concept to themselves (e.g. Chi, Bassok, Lewis, Reimann and Glaser, 1989). The act of creating ISVL movies for other students to view could form an extremely useful learning activity in its own right. Additionally, the ability to communicate and work in teams is becoming considered an ever more important aspect of computer science courses. Team work and the clear presentation of ideas more accurately reflects the current tasks of the industrial programmer than sitting alone carrying out a programming assignment (Dawson, Newsham and Kerridge, 1992). ISVL will allow investigation of how distance education courses can provide more scope for collaborative and group activities within the curriculum. Another envisaged use of ISVL is to provide a framework for debugging communities (Domingue and Mulholland, in press), whereby professional programmers can share knowledge of their particular programming domain.

We intend to explore these technological, theoretical and social issues further within the ISVL project.

6. Summary

The development of ISVL has illustrated the problems and challenges inherent in integrating new technology into a distance education course, whereby improvements need to be made though consistency with existing materials must be maintained. The design of ISVL was motivated by experiences from teaching the course, and detailed empirical evaluation of how students are able to use TPM. Problems with the TPM notation are dealt with by providing a

parallel textual bridging representation, and a reduction in the number of views to be navigated. In order to ensure that students use the environment productively, ISVL encourages individual exploration, and both synchronous and asynchronous communication. The platform independent client-server architecture provides these added resources with only modest bandwidth requirements. The first trial of using ISVL to teach Prolog at a distance is currently under way and we eagerly await the results. The findings will motivate future developments of the course, and will no doubt elicit important lessons as to how new technology can be most effectively used when teaching at a distance.

References

Anderson, J.R. and Reiser, B.J. (1985). *The LISP Tutor*. Byte, 10, 4, 159-175.

Baker, J.E., Cruz, I.F., Liotta, G. and Tamassia, R. (1996). *Algorithm Animation over the World Wide Web*. Proceedings of the International Workshop on Advanced Visual Interfaces, 203-212. ACM Press: New York.

Baecker, R.M. and Sherman, D. (1981). *Sorting Out Sorting*. Narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81. Morgan Kaufmann: Los Altos, CA.

Brayshaw, M. (1994). *Information Management and Visualization for Debugging Logic Programs*. Doctoral Thesis, Human Cognition Research Laboratory, The Open University, Milton Keynes, U.K.

Brayshaw, M. and Eisenstadt, M. (1991). *A Practical Tracer for Prolog*. International Journal of Man-Machine Studies, 35, 597-631.

Brown, M.H. and Najork, M.A. (1996). *Collaborative Active Textbooks: A Web-Based Algorithm Animation System for an Electronic Classroom*. SRC Research Report 142, Digital.

Byrne, M.D., Catrambone, R. and Stasko, J.T. (1996). *Do Algorithm Animations Aid Learning?* Technical Report GIT-GVU-96-18, Graphics, Visualization & Usability Center, Georgia Institute of Technology, U.S.A.

Chi, M.T.H., Bassok, M., Lewis, M. W., Reimann, P. and Glaser, R. (1989). Self Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science*, 13, 145-184.

Davies, S.P. (1994). *Knowledge Restructuring and the Aquisition of Programming Expertise*. International Journal of Human Computer Studies, 40, 703-726.

Dawson, R.J., Newsham, R.W. and Kerridge, R.S. (1992). *Introducing New Software Engineering Graduates to the 'Real World' at the GPT Company*. Software Engineering Journal, 7 (3), 171-176.

Domingue, J. and Mulholland, P. (1997). *Fostering Debugging Communities on the World Wide Web*, Communications of the ACM (Special Issue on Software Visualization: Ed. H. Lieberman), 40 (4), 65-71.

Domingue, J., Price, B. and Eisenstadt, M. (1992) *Viz: A Framework for Describing and Implementing Software Visualization Systems*. In: D. Gilmore D. and R. Winder (Eds.), User Centred Requirements for Software Engineering Environments, Springer-Verlag: London.

Eisenstadt, M. (1984). *A Powerful Prolog Trace Package.* Proceedings of the Sixth European Conference on Artificial Intelligence (ECAI-84), Pisa, Italy.

Eisenstadt, M. (1985). *Tracing and Debugging Prolog Programs by Retrospective Zooming*. In: R. Hawley (Ed.), Artificial Intelligence Programming Environments. Ellis Horwood: Chichester, U.K.

Eisenstadt, M. and Brayshaw, M. (1987). *An Integrated Textbook, Video and Software Environment for Novice and Expert Prolog Programmers.* In: E. Soloway and J. Spohrer (Eds.), *Understanding the Novice Programmer* . Lawrence Erlbaum Associates: Hillsdale, NJ.

Eisenstadt, M. and Brayshaw, M. (1988). *The Transparent Prolog Machine (TPM): An Execution Model and Graphical Debugger for Logic Programming.* Journal of Logic Programming, 5 (4), 277-342.

Eisenstadt, M., Brayshaw, M., Hasemer, T. and Isroff, K. (1996). *Teaching Learning and Collaborating at a Virtual Summer School*. In: A. Dix and R. Beale (Eds.) Remote Cooperation: CSCW Issues for Mobile and Tele-Workers. London: Springer.

Eisenstadt, M., Buckingham Shum, S. and Freeman, A. (1996). *KMi Stadium: Web-Based Audio Visual Interaction as Reusable Organisational Expertise.* Technical Report KMI-TR-31, Knowledge Media Institute, The Open University, U.K.
[http://kmi.open.ac.uk/kmi-abstracts/kmi-tr-31-abstract.html]
[Stadium Project: http://kmi.open.ac.uk/stadium/]

Eisenstadt, M., Dixon, M. and Kriwaczek, F. (1988). *Intensive Prolog*. Academic Press: Milton Keynes, U.K.

Eisenstadt, M., Price, B.A. and Domingue, J. (1992). *Software Visualization: Redressing ITS Fallacies.* In: Proceedings of NATO Advanced Research Workshop on Cognitive Models and Intelligent Environments for Learning Programming, Genova, Italy. Springer-Verlag: Berlin.

Fung, P., Brayshaw, M., du Boulay, B. and Elsom-Cook, M. (1990). *Towards a Taxonomy of Novices' Misconceptions of the Prolog Interpreter.* Instructional Science, 19 (4/5), 311-336.

Karmiloff-Smith, A. (1979). *Micro- and Macro-Developmental Changes in Language Acquisition and Other Representational Systems*. Cognitive Science, 3, 91-118.

Karmiloff-Smith, A. (1992). *Beyond Modularity: A Developmental Perspective on Cognitive Science.* MIT Press/Bradford Books: Cambridge, MA.

Lawrence, A.W., Badre, A.M. and Stasko, J.T. (1994). *Empirically Evaluating the Use of Animations to Teach Algorithms*. IEEE Symposium on Visual Languages, St. Louis, pp. 48-54. IEEE: Los Alamitos, CA.

Mulholland, P. (1995). *A Framework for Describing and Evaluating Software Visualization Systems: A Case-Study in Prolog*. Doctoral Thesis, Knowledge Media Institute, The Open University, U.K. [http://kmi.open.ac.uk/~paulm/usv.html]

Mulholland, P. (in press). *A Principled Approach to the Evaluation of SV: A Case-Study in Prolog.* In: J. Stasko, J. Domingue, M. Brown and B. Price (Eds.), Software Visualization: Programming as a Multi-Media Experience. MIT Press: Cambridge, MA.

Mulholland, P. and Eisenstadt, M. (in press). *Using Software to Teach Computer Programming: Past, Present and Future*. In: J. Stasko, J. Domingue, M. Brown and B. Price (Eds.), Software Visualization: Programming as a Multi-Media Experience. MIT Press: Cambridge, MA.

Pain, H. and Bundy, A. (1987). *What Stories Should We Tell Novice Prolog Programmers?* In: R. Hawley (Ed.), *Artificial Intelligence Programming Environments.* Ellis Horwood: Chichester, U.K.

Petre, M. and Green, T.R.G. (1993). *Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill*. Journal of Visual Languages and Computing, 4, 55-70.

Petre, M., Blackwell, A. and Green, T.R.G. (in press). *Cognitive Questions in Software Visualization*. In: J. Stasko, J. Domingue, M. Brown and B. Price (Eds.), Software Visualization: Programming as a Multi-Media Experience. MIT Press: Cambridge, MA.

Price, B.A., Small, I.S. and Baecker, R.M. (1991). *A Principled Taxonomy of Software Visualization.* Journal of Visual Languages and Computing, 4 (3), 211-266.

Rajan, T. (1986). APT: *A Principled Design for an Animated View of Program Execution for Novice Programmers.* Doctoral Thesis, Human Cognition Research Laboratory, The Open University, U.K..

Stasko, J., Badre, A. and Lewis, C. (1993). *Do Algorithm Animations Assist Learning? An Empirical Study and Analysis*. Proceedings of INTERCHI '93: ACM/IFIP Conference on Human Factors in Computing Systems, 61-66. ACM Press: New York.

Stasko, J., Domingue, J., Brown, M. and Price, B. (in press) (Eds.) *Software Visualization: Programming as a Multi-Media Experience*. MIT Press: Cambridge, MA.

Taylor, C., du Boulay, B. and Patel, M. (1991). *Outline Proposal for a Prolog 'Textual Tree Tracer'* (TTT). Cognitive Science Research Paper 177, School of Cognitive and Computing Sciences, University of Sussex, U.K.

Taylor, J.A. (1988). *Programming in Prolog: An In-Depth Study of the Problems for Beginners Learning to Program in Prolog*. Doctoral Thesis, School of Cognitive and Computing Sciences, University of Sussex, U.K.

Waters, R.C. (1985). *The Programmer's Apprentice: A Session with KBEmacs.* IEEE Transactions on Software Engineering, 11 (11), 1298-1320.